



中山大学 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

《Rust语言与内存安全设计》

第10讲 函数式编程、构建IO项目

课程负责人：陈文清 助理教授
`chenwq95@mail.sysu.edu.cn`

2023年11月1日

目录



- 1 Rust中的函数式编程语言特性**
- 2 习题讲解**
- 3 构建IO项目**

■ 回顾什么是闭包：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数（也可作为返回值）。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包以对其进行评估。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

■ Fn, FnMut和FnOnce特征 (Trait) :

- 闭包从环境中捕获和处理值的方式会影响它实现的特征 (trait) ;
- 闭包将根据其体中的值的处理方式自动实现这三个Fn特质之一、两个或全部三个:
 - FnOnce 适用于**只能调用一次**的闭包。所有闭包至少实现了这个特征，因为所有闭包都可以被调用。
 - FnMut 适用于**不会从闭包中移出捕获值，但可能会更改捕获值的闭包**。这些闭包可以被多次调用。
 - Fn 适用于**不会从其体中移出捕获值，不会更改捕获值的闭包**，以及不从其环境中捕获任何内容的闭包。这些闭包可以被多次调用而不会更改其环境。

示例讲解

// 使用FnOnce的闭包

```
fn execute_fn_once<F: FnOnce()>(f: F) {  
    f();  
    // f(); // 这里会报错, 因为FnOnce只能被调用一次  
}
```

// 使用FnMut的闭包

```
fn execute_fn_mut<F: FnMut()>(mut f: F) {  
    f();  
    f(); // 可以多次调用FnMut闭包  
}
```

// 使用Fn的闭包

```
fn execute_fn<F: Fn()>(f: F) {  
    f();  
    f(); // 可以多次调用Fn闭包  
}
```

```
let mut x: i32 = 42;  
let s1: String = String::from("Hello");  
let s2: String = String::from("Rust");
```

// FnOnce闭包, 只能被调用一次

```
let fn_once_closure: impl Fn() = move || {  
    println!("FnOnce: s1 = {}", s1);  
};
```

// FnMut闭包, 可以被多次调用

```
let mut fn_mut_closure: impl FnMut() = || {  
    x += 1;  
    println!("FnMut: x = {}", x);  
};
```

// Fn闭包, 也可以被多次调用

```
let fn_closure: impl Fn() = || {  
    println!("Fn: s2 = {}", s2);  
};
```



```
let mut x: i32 = 42;
let s1: String = String::from("Hello");
let s2: String = String::from("Rust");

// FnOnce闭包, 只能被调用一次
let fn_once_closure: impl Fn() = move || {
    println!("FnOnce: s1 = {}", s1);
};

// FnMut闭包, 可以被多次调用
let mut fn_mut_closure: impl FnMut() = || {
    x += 1;
    println!("FnMut: x = {}", x);
};

// Fn闭包, 也可以被多次调用
let fn_closure: impl Fn() = || {
    println!("Fn: s2 = {}", s2);
};
```

```
execute_fn_once(fn_once_closure);
execute_fn_once(fn_once_closure); // 这里会报错

execute_fn_mut(&mut fn_mut_closure);
execute_fn_mut(&mut fn_mut_closure);

execute_fn(fn_closure);
execute_fn(fn_closure);
```

迭代器 (Iterator)



什么是迭代器?

迭代器



■ 迭代器 (Iterator)

- 迭代器允许依次对一系列项 (**item**) 执行某些任务
- 迭代器负责:
 - 遍历每个项 (**item**) ;
 - 确定遍历何时完成。
- Rust中的迭代器特点:
 - 懒惰的 (**lazy**) : 这意味着在你调用消耗迭代器的方法之前, 它们不产生任何作用;

迭代器



■ 几种迭代方法（后两种请大家查阅API学习）：

- iter方法：在不可变引用上创建迭代器；
- into_iter方法：创建的迭代器会获得所有权；
- iter_mut方法：迭代可变的引用。

迭代器



■ 几种迭代方法:

➤ into_iter方法: 创建的迭代器会获得所有权;

► Run | Debug

```
fn main() {
```

```
    let v1: Vec<i32> = vec![1, 2, 3];
```

```
    let v1_iter: IntoIter<i32> = v1.into_iter();
```

```
    for v1: i32 in v1_iter {
```

```
        println!("{}", v1);
```

```
    }
```

```
    println!("{:?}", v1);
```

```
}
```

into_iter方法

v1此时不再是*&i32*类型

报错: value borrowed here after move

迭代器



■ 几种迭代方法:

➤ `iter_mut`方法: 迭代可变的引用;

```
fn main() {  
    let mut v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: IterMut<i32> = v1.iter_mut();  
  
    for v1: &mut i32 in v1_iter {  
        *v1 += 1;  
        println!("{}", v1);  
    }  
  
    println!("{:?}", v1);  
}
```

添加**mut**关键词

iter_mut方法

可修改**vector**里的元素

打印结果: [2, 3, 4]

■ 消耗迭代器的方法：

- next方法；
- sum方法：
 - 取得迭代器的所有权；
 - 反复调用next，遍历所有元素；
 - 每次迭代，把当前元素添加到一个总和里，迭代结束返回总和

迭代器



■ 消耗迭代器的方法：

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: Iter<i32> = v1.iter();  
    let total: i32 = v1_iter.sum();  
    assert_eq!(total, 6);  
  
    let v2_iter: Iter<i32> = v1_iter;  
}
```

→ sum方法取得v1_iter的所有权

→ 之后无法再使用v1_iter

■ 产生其他迭代器的方法 ([Methods that Produce Other Iterators](#)) :

- 定义在Iterator Trait上的另一些方法，叫做“迭代器适配器 (Iterator Adaptors)”
 - 把迭代器转换为不同类型的迭代器
- 可以通过链式调用使用多个“迭代器适配器”来进行复杂的操作，可读性较高
- 例如，map：
 - 接收一个闭包；
 - 产生一个新的迭代器

迭代器



■ map:

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: impl Iterator<Item = i32> = v1.iter().map(|x: &i32| x + 1);  
}
```

v1.iter()产生一个基本的迭代器，

map后的一个新的迭代器

迭代器



■ map:

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v2: Vec<_> = v1.iter().map(|x: &i32| x + 1).collect();  
  
    assert_eq!(v2, vec![2, 3, 4]);  
}
```

.collect()方法是一个消耗型适配器，把迭代器耗尽，把结果收集到一个集合中。

■ 迭代器适配器:

- 通常结合闭包使用
- 一种重要的用法是闭包里捕获外部“环境”中的变量
- 以filter方法为例

迭代器



■ 迭代器适配器:

- 通常结合闭包使用

```
fn main() {  
    // 创建一个包含数字的向量  
    let numbers: Vec<i32> = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
  
    let double_numbers: Vec<i32> = numbers.iter().map(|&x: i32| x * 2).collect();  
    println!("原始: {:?}", numbers);  
    println!("翻倍后: {:?}", double_numbers);  
  
    // 使用filter迭代器方法筛选出偶数  
    let even_numbers: Vec<i32> = numbers.iter().filter(|&&x: i32| x % 2 == 0).cloned().collect();  
    // 输出筛选后的结果  
    println!("偶数: {:?}", even_numbers);  
}
```

迭代器














■ 迭代器适配器:

➤ 通常结合闭包使用

```
fn main() {  
    // 创建一个包含数字的向量  
    let numbers: Vec<i32> = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
  
    let double_numbers: Vec<i32> = numbers.iter().map(|&x: i32| x * 2).collect();  
    println!("原始: {:?}", numbers);  
    println!("翻倍后: {:?}", double_numbers);  
  
    // 使用filter迭代器方法筛选出偶数  
    //let even_numbers: Vec<i32> = numbers.iter().filter(|&x| x % 2 == 0).cloned().collect();  
    let even_numbers: Vec<i32> = numbers.into_iter().filter(|&x: i32| x % 2 == 0).collect();  
    // 输出筛选后的结果  
    println!("偶数: {:?}", even_numbers);  
}
```

注意.into_iter的使用

-  iterators1.rs
-  iterators2.rs
-  iterators3.rs
-  primitive_types4.rs
-  primitive_types5.rs
-  primitive_types6.rs
-  strings3.rs
-  strings4.rs
-  variables4.rs
-  variables5.rs
-  variables6.rs

第二次作业



题目：几何形状管理程序（考察Struct、Trait、Generic的用法）

要求：

1. 创建一个名为Shape的Trait，其中包括以下方法：
 1. `area(&self) -> f64`：计算几何形状的面积。
 2. `perimeter(&self) -> f64`：计算几何形状的周长。
2. 创建三个Struct，分别代表以下几何形状，每个Struct都必须实现Shape Trait：
 1. 矩形（Rectangle）：包含长度和宽度。
 2. 圆形（Circle）：包含半径。
 3. 三角形（Triangle）：包含三条边的长度。
3. 创建一个泛型函数`print_shape_info<T: Shape>(shape: T)`，它接受任何实现了Shape Trait的几何形状，然后打印该几何形状的类型、面积和周长。
4. 在main函数中，创建至少一个矩形、一个圆形和一个三角形的实例，并使用`print_shape_info`函数分别输出它们的信息。

第二次作业



题目：几何形状管理程序（考察Struct、Trait、Generic的用法）

要求：

1. 创建一个名为Shape的Trait，其中包括以下方法：

1. `area(&self) -> f64`: 计算几何形状的面积。
2. `perimeter(&self) -> f64`: 计算几何形状的周长。

2. 创建三个Struct，分别代表以下几何形状，每个Struct都必须实现Shape Trait:

1. 矩形（Rectangle）：包含长度和宽度。
2. 圆形（Circle）：包含半径。
3. 三角形（Triangle）：包含三条边的长度。

3. 创建一个泛型函数`print_shape_info<T: Shape>(shape: T)`，它接受任何实现了Shape Trait的几何形状，然后打印该几何形状的类型、面积和周长。

4. 在main函数中，创建至少一个矩形、一个圆形和一个三角形的实例，并使用`print_shape_info`函数分别输出它们的信息。

提示：

- 在Shape Trait中，你可以使用关联类型来定义几何形状的属性，例如面积和周长的类型。这样，每个实现Trait的类型可以定义自己的关联类型。
- 在main函数中，可以使用泛型函数`print_shape_info`来减少代码重复。

这个作业将考察学生对Trait、Struct、以及泛型的理解和运用。学生需要创建自定义的Struct，并确保它们实现了Trait中定义的方法。同时，他们需要编写一个泛型函数来处理不同类型的几何形状，并灵活地使用Trait来计算面积和周长。这有助于加强对Rust中泛型和Trait系统的理解。

第二次作业



2023秋rust课程第二次作业收集

截止时间：2023-11-07 23:59

提交地址：<https://send2me.cn/0CgN5q-f/Qmyaitg1pJl1dw>

构建I/O项目



- 1 接收命令行参数
- 2 读取文件
- 3 重构（模块化和错误处理）
- 4 测试驱动开发（Test-Driven Development）
- 5 结合环境变量
- 6 将错误消息写入标准错误而不是标准输出

■ 构建一个I/O项目：命令程序

- Rust 的速度、安全性、单一二进制输出和跨平台支持使其成为**创建命令行工具**的理想语言。
- 我们将制作经典命令行搜索工具 `grep`（全局搜索和打印）。在最简单的用例中，`grep` 在指定文件中搜索指定字符串。为此，`grep` 将文件路径和字符串作为其参数。然后它读取文件，在该文件中找到包含字符串参数的行，并打印这些行。

```
root@Linux-world:~# grep linuxtechi /etc/passwd
linuxtechi:x:1000:1000:linuxtechi,,,:/home/linuxtechi:/bin/bash
root@Linux-world:~#
```

■ 需要用到过去讲解的内容:

- 组织代码（关于模块的知识）
- 使用向量和字符串（集合collections）
- 处理错误在适当的地方使用特征和生命周期（错误处理）
- 编写测试

1. 接收命令行参数



- 创建一个新的项目minigrep（为了区分系统命令grep）：

```
$ cargo new minigrep
    Created binary (application) `minigrep` project
$ cd minigrep
```

- 第一个任务是让 minigrep 接受它的两个命令行参数：文件路径和要搜索的字符串。也就是说，我们希望能够使用 cargo run 运行我们的程序，包含两个参数：一个要搜索的字符串，以及一个要搜索的文件的文件的路径，就像这样

```
$ cargo run -- searchstring example-filename.txt
```

1. 接收命令行参数



■ 读取命令行的参数值：

- 为了使 minigrep 能够读取我们传递给它的命令行参数，我们需要 Rust 标准库中提供的 `std::env::args` 函数。此函数返回传递给 minigrep 的命令行参数的迭代器。
- 迭代器产生一系列值，我们可以调用迭代器的 `collect` 方法将其变成一个集合，例如向量，包含迭代器产生的所有元素。

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

1. 接收命令行参数



■ 读取命令行的参数值：

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

- 首先，我们使用 `use` 语句将 `std::env` 模块引入作用域，以便我们可以使用它的 `args` 函数。请注意，`std::env::args` 函数嵌套在两层模块中。
- **注意代码规范：** 如果所需函数嵌套在多个模块中，我们选择将父模块而不是函数引入范围。通过这样做，我们可以轻松地使用 `std::env` 中的其他函数。
- **注意代码规范：** 它也比添加 `use std::env::args` 然后仅使用 `args` 调用函数更明确，因为 `args` 可能很容易被误认为是当前模块中定义的函数。

1. 接收命令行参数



■ 保持命令行的参数值到变量中：

目前程序能够访问指定为命令行参数的值。现在我们需要将两个参数的值保存在变量中，以便我们可以在程序的其余部分使用这些值。

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

程序的名称在 `args[0]` 处占据了向量中的第一个值，因此我们从索引 1 开始。

1. 接收命令行参数



■ 保持命令行的参数

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

```
$ cargo run -- test sample.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0s
Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

2. 读取文件



■ 目标场景：读取 file_path 参数中指定的文件

首先，我们需要一个示例文件来对其进行测试。在项目的根级别创建一个名为 poem.txt 的文件，该文件包含多行的少量文本和一些重复的单词。Emily Dickinson的诗

Filename: poem.txt

```
I'm nobody! Who are you?  
Are you nobody, too?  
Then there's a pair of us - don't tell!  
They'd banish us, you know.
```

```
How dreary to be somebody!  
How public, like a frog  
To tell your name the livelong day  
To an admiring bog!
```


2. 读取文件



■ 目标场景：读取 file_path 参数中指定的文件

首先，我们使用 `use` 语句引入标准库的相关部分：我们需要 `std::fs` 来处理文件。

在 `main` 中，新语句 `fs::read_to_string` 获取 `file_path`，打开该文件，并返回文件内容的 `std::io::Result<String>`。

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

2. 读取文件



■ 目标场景：读取 file_path 参数中指定的文件

让我们使用任意字符串作为第一个命令行参数（因为我们还没有实现搜索部分）和 poem.txt 文件作为第二个参数来运行这段代码：

```
$ cargo run -- the poem.txt
  Compiling minigrep v0.1.0 (file:///projects/minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.0s
  Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

2. 读取文件



- 目标场景：读取 file_path 参数中指定的文件

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

目前代码有一些缺陷。主函数有多个职责：一般来说，如果每个函数只负责一个idea，函数会更清晰，也更容易维护。另一个问题是我们没有尽我们所能处理错误。该程序仍然很小，所以这些缺陷不是什么大问题，但是随着程序的增长，将很难彻底修复它们。在开发程序时尽早开始重构是一种很好的做法，因为重构少量代码要容易得多。



中山大學
SUN YAT-SEN UNIVERSITY

Q & A

Thanks!